Chapter 6

Configuration Management

So far we have discussed processes (Chapter 3) and how processes can be used to solve software engineering problems, can be used to reduce project risk and increase the likelihood of success by understanding what went well last time and building it into a process.

We have discussed people and teams (Chapter 4) and what motivates people, forges them into performing teams and the methods of power and governance. People are essential to the successful completion of projects and must work with the process to achieve the goals of the project. Essentially, the process provides the template for the steps that the team will take to solve the software engineering problem and philosophy for how the team will work together in solving the problem.

We have discussed the project plan (Chapter 5) and the development of a project schedule from a process by creating a work breakdown structure, determining the dependencies between tasks, and determining the resources needed to complete each task. We also saw how to find the critical path and how the critical path and critical activities impact upon project completion times.

The final controlling discipline is *software configuration management*, which is the discipline of ensuring that all artifacts contributing to the goals of the project are consistent with each other and their own goals. This is especially important under situations where changes are occurring frequently.

6.1 Configurations and their Management

Software projects generate a large number of different types of artifacts. For example, in a typical mode driven process[1] using UML 2.0 we might find:

- use-cases relevant to the problem;
- class diagrams, collaboration diagrams, activity diagrams or state charts that model the problem domain;
- class diagrams, collaboration diagrams, activity diagrams or state charts that specify the design[2];
- all of the code modules that make up the code base for the project[3];
- the test cases, testing reports and testing scripts; and
- all documents relevant to the project such as the project management plan, test plan or configuration management plan.

[1] A model driven process is one in which the focus of the analysis is on creating models that are much closer to domain concepts rather than computing concepts.

[2] **Note** the domain analysis model in a model driven approach to software engineering may well be different to design models. Domain models must focus on domain objects and domain actions, while design models often focus on the data structures and algorithms used to implement the domain model.

[3] Recall that there are over 1600 modules in both Linux and Mozilla.

There are dependencies between all of these artifacts. For example, a code module may depend on a design element such as a class diagram or state chart, as well as on a design element such as a design class diagram. In turn these may depend on a combination of textual requirements, use-cases and analysis classes.

Definition 6.1. *Configuration* The sum total of all the artifacts, their current state and the dependencies between them is called the *configuration*.

The problem is change! If we make a change to an artifact or work product then we may impact all of that artifact's dependencies. If we are not careful then changes to artifacts may leave the configuration in an *inconsistent state*. For example, a change to the requirement impacts will have an impact on the system design and all of the code modules that depend on the design. Also, the testing plan, test cases and testing scripts for the code will also be impacted. The danger is that we may change one module without changing one of its dependent modules leaving the configuration inconsistent.

The aim of configuration management is to establish processes and set up repositories to manage change properly without losing overall consistency. The kinds of questions that configuration management addresses are the following:

- How do we manage requests for change?
- What and where are the software components?
- What is the status of each software component?
- How does a change to one component affect others?
- How do we resolve conflicting changes?
- How do we maintain multiple versions?
- How do we keep the system up to date?

6.2 Configuration Management Processes

Software configuration management (SCM) processes typically have the following four aims [Pre09].

1. to identify all items that collectively will make up the configuration;
2. to manage changes to one or more of these items so that the collection remains consistent;
3. to manage different versions of the product; and
4. to assure software quality as the configuration evolves over time.

The five key configuration management tasks that meet these aims are:

1. **identification** — where the configuration items necessary for the project are identified;
2. **version control** — where processes and tools are chosen to manage the different versions of configuration items as they are developed;
3. **change control** — where changes to that affect more than just one configuration item are managed;
4. **configuration auditing** — where the consistency of the configuration is checked; and
5. **configuration reporting** — where the status of configuration items is reported.

Identifying Configuration Items

The configuration consists of the entire set of artifacts and artifacts produced in the project. These are called *configuration items*. Some of these configuration items are *basic objects*, while others are *aggregate*, and still others are *derived*. Aggregate objects are composed of atomic objects or other aggregate objects. Derived objects are generated from atomic and aggregate objects by tools.

For example, we may choose classes to be basic objects stored in files in a repository. The program is the aggregate object also stored in a file but dependent on the objects of which it is composed. When we compile the program we generate a derived object — the executable — which is also stored in a file but is derived from the program and object files.

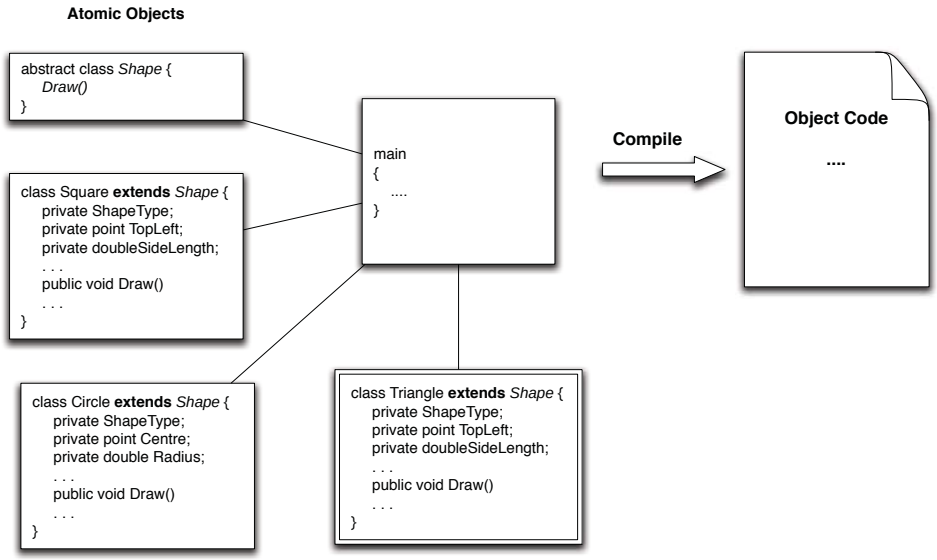

Figure 6.1: A fragment of a configuration that includes *atomic objects* (classes Shape, Triangle, Circle and Square), *aggregate objects* (the main program), and *derived objects* (the object code). The tool used to generate the object code from the program files is a compiler. The program is dependent on the classes in the class files.

The aim in identification is to determine what configuration items will be produced in the project and how they will be managed. A typical list of software configuration items may include (but is not limited to):

- requirements specifications, requirements models, sections of the requirements specification, and individual requirements;
- use-cases;
- design models, design documents, design elements, and class designs;
- source code modules;
- object code modules;
- release modules;
- software tools;
- test drivers and stubs, and test scripts; and
- documents or sections of documents associated with the project.

Version Control

Version control refers to managing the different versions of all the configuration objects in the project. Version management is typically done using version control systems that consist of:

1. a repository for storing configuration items;
2. version management functions that allow software engineers to create and track versions, and roll the system back to previous versions if necessary; and
3. ideally a *make* facility that allows engineers to collect all of the configuration objects for a particular target together and to build that target.

Notice that, from a management and planning perspective there is a choice to make at this point. We need to answer the following questions.

> What objects do we need to track and at what level of detail (or granularity) will we need to track objects?

For example, if we are considering just the requirements process then do we track our requirements document in its entirety, or do we track each section and model? From a management perspective, if we choose the former and just track the document then all changes are at the level of the document and — as managers who are interested in progress — all we can track is the completion of the document. On the other hand if we choose to put the sections of the requirements under version control then — as project managers interested in progress — we can track the progress of sections as well as the document as a whole. The trade off is that there is an overhead in tracking at a finer level of detail and additional computing resources will be needed to keep track of the extra information.

All SCM information should be maintained in a *repository* or *configuration database*. Be wary of tools such as CVS as they may only store information on versions and not the dependencies between different configurations items. The following terminology typically applies to version control systems.

Version — An instance of a model, document, code, or other configuration item which is functionally distinct in some way from other system instances.

Variant — An instance of a system which is functionally identical but non-functionally distinct from other instances of a system.

Release — An instance of a system which is distributed to users outside of the development team.

Once we have a configuration item under version control then it will have a *derivation history*. This is a record of changes applied to a configuration object. Ideally, each change should record:

- the change made;
- the rationale for the change;
- who made the change; and
- when it was implemented.

The information about a change may be included as a comment in code. If a standard prologue style of commentary is used for the derivation history in the code then tools can process this history automatically and insert it into the derivation of a configuration object in the version control repository.

A common method of tracking versions in a repository is through *version numbering*, in which versions are numbered and tagged. In this case the actual numbers typically have no meaning on their own but take meaning from the overall management of configuration objects. For example, major version numbers represent reviewed versions while minor version numbers represent small un-reviewed changes. Figure 6.3 shows to different interpretations of version numbers.

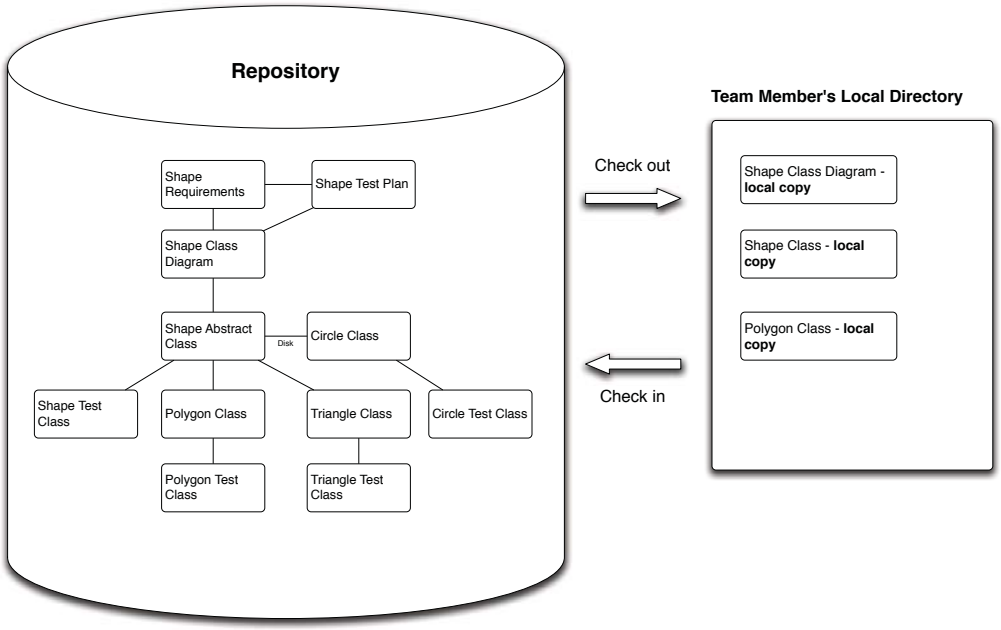

Figure 6.2: A simple version control repository. Items are checked into the repository and local working copies are checked out of the repository.

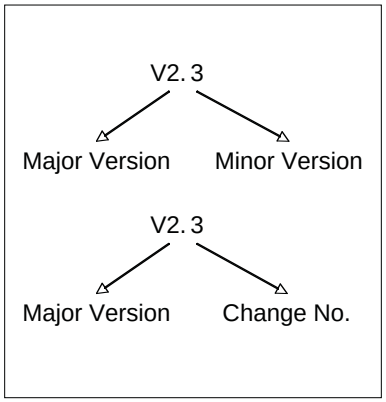

Figure 6.3: Two different uses of the major and minor version numbers. The interpretation of these depends on the configuration management plan.

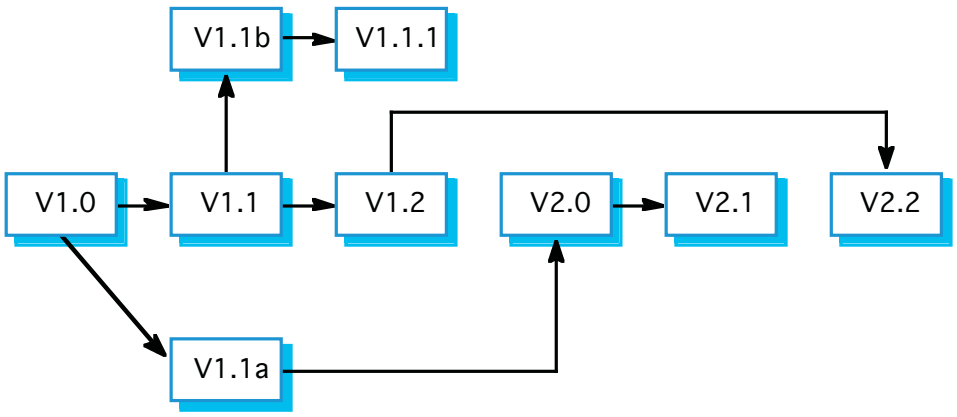

Figure 6.4: An example of a derivation structure for a project using version numbering to mark branches and merges.

The *derivation history* of a configuration item is the sequence of changes, tracked by version numbers, that the configuration item went through. If we are using a version numbering style of tracking then the derivation history may appear as in Figure 6.4.

Another popular method of tracking versions in a repository is through *attributes*. Examples of attributes include: Date; Creator; Programming Language; Customer; and Status. Ideally, attributes should uniquely identify the version of the configuration object but this is not always the case. Attributes however, can be much more flexible for searching and retrieving configuration objects because we do not need to remember how the version number corresponds to the attributes of the configuration object. The set of attributes have to be chosen carefully so that all versions can be uniquely identified. In practice, a version also needs an associated name for easy reference and typically both numbered and attribute style of version tracking are used.

Change Control

Once the plan is set and the project is set in motion it may seem that there is little left to do except to *steer the ship*. Unfortunately, change that effects more than just one configuration item and more than one person often happens. It can be to take advantage of new technologies or opportunities, or because as we learn more about what the product that we are building and we want to change our plans in order to reflect our new understanding. Just some of the types of changes are given in Table 6.1.

Discrepancies	**Requested Changes**
• Refining, extending or correcting problems in requirements;	• Enhancements and additional features;
• Modifying, extending or correcting the design;	• Changes to requirements;
• Changing technology.	• Refinement of existing requirements;
	• Changes to design.

Table 6.1: Some of the motivations for making changes to a project.

Changes to the project often follow the sequence of steps in Figure 6.5.

The *change management plan* is developed as part of an overall configuration management plan specifically to control these changes to the configuration. Changes must be made in a way that allows everyone on the project team to find out exactly what changes need to be made, what they need to do to affect the change, why the change is being made, and how it will impact them. More importantly, in distributed control structures, some changes may need to

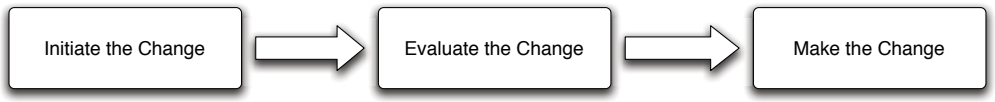

Figure 6.5: Three simple steps to making a change.

be carefully negotiated so that everyone understands the need for the change and supports it. Table 6.2 gives some questions that need to be considered when making a change.

Element	Impact on the Process
Initiate the Change	Why is the change being made? How are changes requested and tracked? What information will be needed to evaluate the change? How will the change be evaluated?
Evaluate the Change	How will the change affect the configuration? Which artifacts need to change and what are their dependencies? What are the benefits of the change? What are the costs of the change? Do the benefits of the change outweigh the costs of the change? Who will be effected by the change?
Making the Change	Who will put the change into effect? How will the change be managed? How will other people working on the project understand the change? How will they be notified of the change? How will people working on the project know when the change is completed?

Table 6.2: Elements of a change control plan.

Table 6.3 summarises just some of the factors that are considered when evaluating a change.

• Size	• Criticality of the area involved
• Complexity	• Politics from customer or marketing
• Date required	• Approved changes already in progress
• CPU and memory impact	• Resources in terms of skills, hardware, or system availability
• Cost	• Impact on current and subsequent work
• Test requirements	• Is there an alternative?

Table 6.3: Some of the motivations for making changes to a project.

Baselines

The question of "how will people working on the project know when the change is completed?" is part of a more general problem. *How do we know when a configuration object is ready to be used*? Typically when we work on a specification or design in a small team then changes are made quickly and we do not worry about the formal sorts of change processes described above. The aim is to get a draft of a document or model out for others to try or to produce code that is ready to review and test.

However, once the task of getting the artifact built and reviewed is done it is available for others to use. Once other people start using the artifact for their work changes are more difficult to make because more people are affected. For example, imagine developing a system that uses a component built by another team in your organisation. One day, your system fails to compile because the interface to the component has changed in the latest version. It then takes weeks for your team to update your system to use the new version of the component. If the organisation insisted that these changes were monitored correctly, you may have learned about this early enough to create a new branch of development to handle the change properly.

An important idea in configuration management is to differentiate all of the configuration items that are *stable* and can be used by others from those that are *unstable* and are changing too frequently to be used. The idea is often tied up with the idea of a *baseline*.

Definition 6.2. *Baseline* A baseline is an artifact that is *stable*. That is, it has been formally reviewed and agreed upon, that is now ready for use in future development, and can only be changed through formal change management procedures.

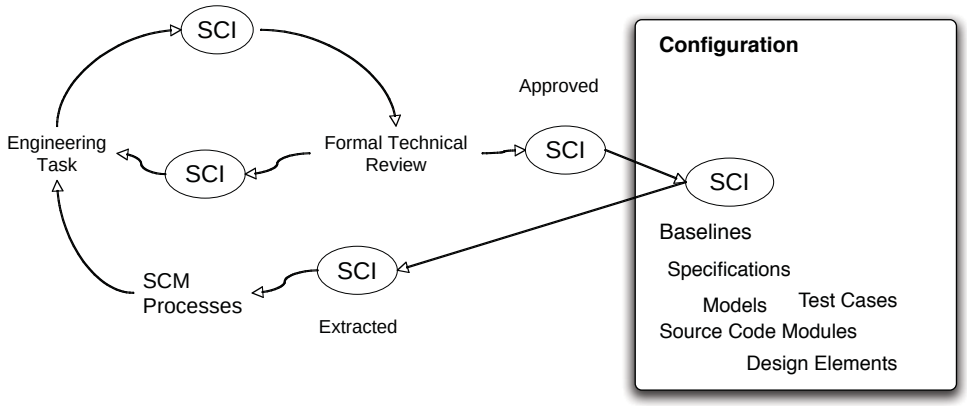

Figure 6.6: Managing through baselines.

The process looks something like the picture in Figure 6.6 [Pre09]. In this figure, we have a software configuration item **SCI**. First we extract it from the repository and then perform our engineering tasks using the **SCI**. We may make frequent changes to the **SCI** while doing this but at some point the engineering task concludes and we are left with a modified **SCI** ready to be baselined. If the **SCI** passes the formal review then it can be added to the stable configuration items already in the repository. If it does not pass the review then it must be reworked until it does. Other project team members only work with the baselined stable **SCI**s. Once baselined any further changes need to go through the formal change process.

We can have many different baselines in a project and its not uncommon for teams to be working with a requirements package at baseline 3.0 and a design at baseline 2.0. The key point is that the designs are stable (not changing) and are consistent with the stable requirements!

6.3 Auditing and Status Reporting

Identification, version control and change control help to maintain a good measure of order to what, can be, quite a fluid situation. *Configuration audits* complement the other configuration management activities by assuring that what is in the repository is actually consistent and that all of the changes have been made properly.

Auditing and *status reporting* are common ways on large projects to keep track of the status of the repository. The idea is to review the configuration objects for consistency with other configuration objects, to find any omissions or to look for potential side effects. The types of questions typical as part of a configuration audit are sketched in Table 6.4.

Status reporting can take many forms, but most commonly the aim is to report on the status of the configuration items of interest and the baselines that have been achieved. Configuration items in a repository may be in one of a number of states and the state may be used as part of the status report. For example, we may have a design element that is in one of the states: *not-initiated*, *initial-work*, *modified*, *approved*, *baselined*. The status report can easily just give the state of the design element which can then be compared with the project plan to see if any action needs to be taken.

• Have the changes requested and approved been made?	• Have any additional changes required by a request been made?
• Have the configuration objects that were changed passed their quality assurance tests?	• Do the objects in the configuration meet the required external standards?
• Do the attributes of the configuration item match the change?	• Does every configuration item have appropriate change logs?

Table 6.4: Questions for a configuration audit.